24 | Go语言编译器：把它当作教科书吧

2020-07-27 宫文学

编译原理实战课 进入课程 >

讲述：宫文学
时长 19:49 大小 18.16M

你好，我是宫文学。今天这一讲，我来带你研究一下 Go 语言自带的编译器，它可以被简称为 gc。

我之所以要来带你研究 Go 语言的编译器，一方面是因为 Go 现在确实非常流行，很多云端服务都用 Go 开发，Docker 项目更是巩固了 Go 语言的地位；另一方面，我希望你能把它当成编译原理的教学参考书来使用。这是因为：

> Go 语言的编译器完全用 Go 语言本身来实现，它完全实现了从前端到后端的所有工作，而不像 Java 要分成多个编译器来实现不同的功能模块，不像 Python 缺少了后端，不像 Julia 用了太多的语言。所以你研究它所采用的编译技术会更方便。

Go 编译器里基本上使用的都是经典的算法：经典的递归下降算法、经典的 SSA 格式的 IR 和 CFG、经典的优化算法、经典的 Lower 和代码生成，因此你可以通过一个编译器就把这些算法都贯穿起来。

除了编译器，你还可以学习到一门语言的其他构成部分的实现思路，包括运行时（垃圾收集器、并发调度机制等）、标准库和工具链，甚至连链接器都是用 Go 语言自己实现的，从而对实现一门语言所需要做的工作有更完整的认识。

最后，Go 语言的实现继承了从 Unix 系统以来形成的一些良好的设计哲学，因为 Go 语言的核心设计者都是为 Unix 的发展，做出过重要贡献的极客。因此了解了 Go 语言编译器的实现机制，会提高你的软件设计品味。

扩展：每种语言都有它的个性，而这个个性跟语言设计者的背景密切相关。Go 语言的核心设计者，是 Unix 领域的极客，包括 Unix 的创始人和 C 语言的共同发明人之一，Ken Tompson。Rob Pike 也是 Unix 的核心作者。

Go 语言的作者们显然希望新的语言体现出他们的设计哲学和口味。比如，致力于像 Unix 那样的简洁和优雅，并且致力于让 Go 再次成为一款经典作品。

所以，在已经研究了多个高级语言的编译器之后，我们可以拿 Go 语言的编译器，把整个编译过程再重新梳理和印证一遍。

好了，现在就开始我们今天探索的旅途吧。

首先，我们来看看 Go 语言编译器的前端。

重要提示：照例，你要下载 Go 语言的源代码，本讲采用的是 1.14.2 版本。并且，你最好使用一个 IDE，便于跟踪调试编译器的执行过程。

Go 的源代码中附带的介绍编译器的文档，写得很好、很清晰，你可以参考一下。

词法分析和语法分析

Go 的编译器的词法分析和语法分析功能的实现，是在 cmd/compile/internal/syntax 目录下。

词法分析器是 scanner.go。其实大部分编程语言的词法分析器的算法，都已经很标准了，我们在Java 编译器里就曾经分析过。甚至它们处理标识符和关键字的方式也都一致，都是先作为标识符识别出来，然后再查表挑出关键字来。Go 的词法分析器并没有像 V8 那样在不遗余力地压榨性能，它跟你平常编码的方式是很一致的，非常容易阅读。

语法分析器是 parser.go。它是一个标准的手写的递归下降算法。在解析二元表达式的时候，Go 的语法分析器也是采用了运算符优先级算法，这个已经是我们第 N 次见到这个算法了，所以你一定要掌握！不过，每个编译器的实现都不大一样，而 Go 的实现方式相当的简洁，你可以去自己看一下，或者用调试器来跟踪一下它的执行过程。

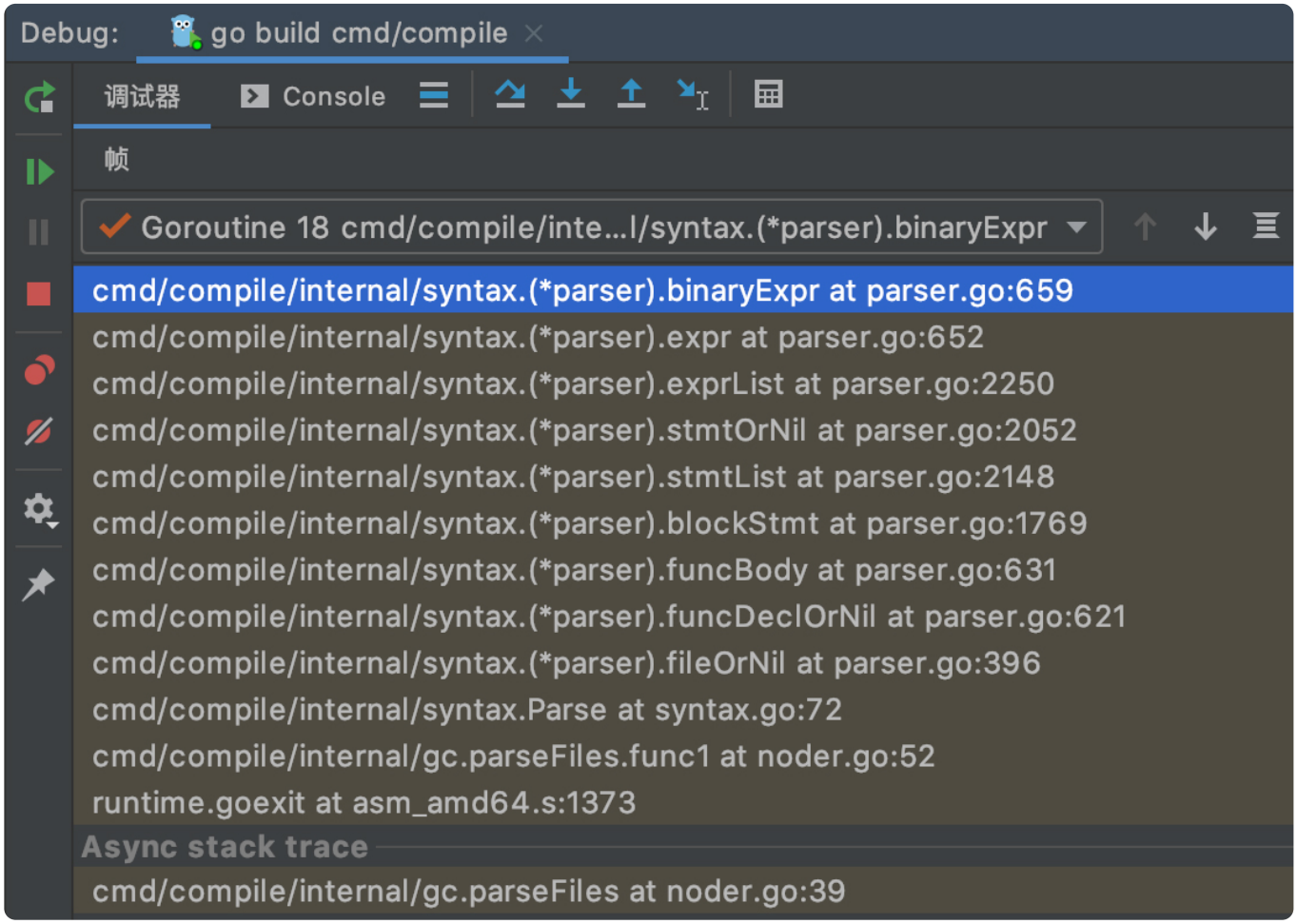


图 1：用 IDE 工具 Goland 跟踪调试编译过程

Go 的 AST 的节点，是在 nodes.go 中定义的，它异常简洁，可以说简洁得让你惊讶。你可以欣赏一下。

Go 的语法分析器还有一个很有特色的地方，就是对错误的处理。它在处理编译错误时，有一个原则，就是不要遇到一个错误就停止编译，而是要尽可能跳过当前这个出错的地方，

继续往下编译，这样可以一次多报几个语法错误。

parser.go 的处理方式是，当语法分析器在处理某个产生式的时候，如果发现了错误，那就记录下这个错误，并且往下跳过一些 Token，直到找到一个 Token 是属于这个产生式的 Follow 集合的。这个时候编译器就认为找到了这个产生式的结尾。这样分析器就可以跳过这个语法单元，继续处理下面的语法单元。

比如，在解析函数声明语句时，如果 Go 的语法分析器没有找到函数名称，就报错“expecting name or (”，然后往后找到“{”或者“;”，这样就跳过了函数名称的声明部分，继续去编译后面的函数体部分。

在 cmd/compile/internal/syntax 目录下，还有词法分析器和语法分析器的测试程序，你可以去运行测试一下。

最后，如果你还想对 Go 语言的语法分析有更加深入地了解，我建议你去阅读一下Go 语言的规范，它里面对于每个语法单元，都有 EBNF 格式的语法规则定义，比如对语句的定义。你通过看代码、看语言规范，积累语法规则的第一手经验，以后再看到一段程序，你的脑子里就能反映出它的语法规则，并且能随手画出 AST 了，这是你学习编译原理需要建立的硬功夫。比如说，这里我节选了一段 Go 语言的规范中针对语句的部分语法规则。

复制代码

```
Statement =
  Declaration | LabeledStmt | SimpleStmt |
  GoStmt | ReturnStmt | BreakStmt | ContinueStmt | GotoStmt |
  FallthroughStmt | Block | IfStmt | SwitchStmt | SelectStmt |
    ForStmt | DeferStmt .

SimpleStmt = EmptyStmt | ExpressionStmt | SendStmt | IncDecStmt |
    Assignment | ShortVarDecl .
```

好，在了解了 Go 语言编译器的语法分析工作以后，接下来，我们再来看看它的语义分析阶段。

语义分析（类型检查和 AST 变换）

语义分析的程序，是在 cmd/compile/internal/gc 目录下（注意，gc 的意思是 Go Compiler，不是垃圾收集的意思）。在入口代码 main.go 中，你能看到整个编译过程的主干步骤。

语义分析的主要程序是在typecheck.go中。**这里你要注意，**不要被“typecheck”的名称所误导，它其实不仅是做类型检查，还做了名称消解（Name Resolution）和类型推导。

你已经知道，名称消解算法的特点，是分阶段完成。举个例子，在给表达式“a=b”中的变量 b 做引用消解之前，编译器必须先处理完 b 的定义，比如“var b Person”，这样才知道符号 b 指的是一个 Person 对象。

另外，在前面学习Java 编译器的时候，你已经知道，对方法体中的本地变量的消解，必须放在最后，才能保证变量的使用总是引用到在它前面的变量声明。Go 的编译器也是采用了相同的实现思路，你可以借此再回顾一下这个知识点，加深认识。

在语义分析阶段，Go 的编译器还做了一些 AST 变换的工作。其中就有内联优化和逃逸分析这两项工作。在我们之前解析的编译器当中，这两项工作都是基于专门做优化的 IR（比如 Sea of Nodes）来做的，而在 Go 的编译器里，却可以基于 AST 来做这两项优化。你看，是不是真实世界中的编译器，才能让你如此开阔眼界？

你可以用“-m”参数来编译程序，它会打印出与内联和逃逸方面有关的优化。你可以带上多个“-m”参数，打印出嵌套层次更深的算法步骤的决策。

复制代码

```
1 go build -gcflags '-m -m' hello.go
```

好了，现在我们借 gc 编译器，又复习了一遍语义分析中的一些关键知识点：名称消解算法要分阶段，在语义分析阶段会对 AST 做一些变换。我们继续来看 gc 编译器下一步的处理工作。

生成 SSA 格式的 IR

gc 编译器在做完语义分析以后，下一步就是生成 IR 了。并且，gc 的 IR 也是 SSA 格式的。你可以通过 gc，来进一步了解如何生成和处理 SSA 格式的 IR。

好，首先，我们来看看 Go 语言的 IR 是什么样子的。针对下面的示例代码 foo.go，我们来看下它对应的 SSA 格式的 IR：

复制代码

```
package main

func Foo(a int) int {
    var b int
    if (a > 10) {
        b = a
    } else {
        b = 10
    }
    return b
}
```

在命令行中输入下面的命令，让 gc 打印出为 foo 函数生成的 IR。在当前目录下，你能看到一个 ssa.html 文件，你可以在浏览器中打开它。

复制代码

```
GOSSAFUNC=Foo go build -gcflags '-S' foo.go
```

在这个文件当中，你能看到编译器对 IR 做了多步的处理，也能看到每次处理后所生成的 IR。

gc 的 IR 是基于控制流图（CFG）的。一个函数会被分成多个基本块，基本块中包含了一行行的指令。点击某个变量，你能看到它的定义和使用情况（def-use 链，图中显示成绿色）。你还能看到，图中灰色的变量，根据定义和使用关系，会发现它们没有被使用，所以是死代码，可以删除。

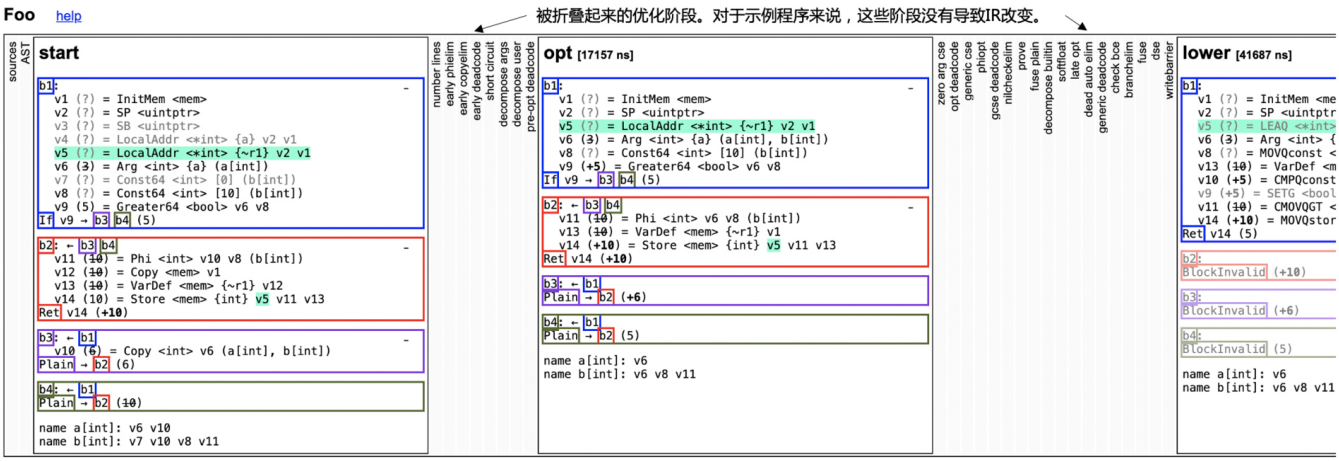


图 2：foo 示例程序各个处理阶段的 IR

针对第一个阶段（Start 阶段），我来给你解释一下每行指令的含义（可参考 genericOps.go），帮助你了解 Go 语言的 IR 的设计特点。

指令	含义
b1:	b1基本块，是函数的入口块
v1 (?) = InitMem <mem>	函数使用的内存
v2 (?) = SP <uintptr>	栈指针
v3 (?) = SB <uintptr>	栈底指针
v4 (?) = LocalAddr <*int> {a} v2 v1	本地变量的地址（在内存v1中，相对于栈指针v2的偏移量）
v5 (?) = LocalAddr <*int> {~r1} v2 v1	返回值的地址（~r1代表第一个返回值）
v6 (3) = Arg <int> {a} (a[int])	把参数a的值赋给v6
v7 (?) = Const64 <int> [0] (b[int])	给v7赋初始值0 v7也就是本地变量b
v8 (?) = Const64 <int> [10] (b[int])	给v8赋值10
v9 (5) = Greater64 <bool> v6 v8	比较v6和v8的大小，赋给v9 也就是计算a>10的值
If v9 → b3 b4 (5)	基于v9的值做跳转 v9为ture，跳转到b3，否则到b4
b2: ← b3 b4	b2基本块，它是函数的退出块
v11 (10) = Phi <int> v10 v8 (b[int])	phi节点 根据控制流中的前序基本块来确定v11的值：如果前序基本块是b3，则取v10的值；如果是b4，则取v8的值
v12 (10) = Copy <mem> v1	把v1的值拷贝到v12
v13 (10) = VarDef <mem> {~r1} v12	在内存v12中定义一个新变量~r1
v14 (10) = Store <mem> {int} v5 v11 v13	把v11的值写到v5，所使用的内存是v13
Ret v14 (+10)	返回v14
b3: ← b1	b3基本块
v10 (6) = Copy <int> v6 (a[int], b[int])	把v6的值拷贝给v10
Plain → b2 (6)	跳转到b2
b4: ← b1	基本块b4
Plain → b2 (10)	跳转到b2

你可以参考代码库中介绍 SSA 的文档，里面介绍了 Go 的 SSA 的几个主要概念。

下面我来给你解读一下。

首先是值（Value）。Value 是 SSA 的最主要构造单元，它可以定义一次、使用多次。在定义一个 Value 的时候，需要一个标识符（ID）作为名称、产生该 Value 的操作码（Op）、一个类型（Type，就是代码中 <> 里面的值），以及一些参数。

操作码有两类。一类是机器无关的，其定义在genericOps.go中；一类是机器相关的，它是面向特定的 CPU 架构的，其定义在 XXXOps.go 中。比如，AMD64Ops.go中是针对 AMD64 架构 CPU 的操作码信息。

在做 Lower 处理时，编译器会把机器无关的操作码转换为机器相关的操作码，有利于后序生成目标代码。机器无关的优化和机器相关的优化，分别作用在采用这两类不同操作码的 IR 上。

Value 的类型信息，通常就是 Go 语言中的类型。但有几个类型是只会在 SSA 中用到的特殊类型，就像上面语句中的，即内存 (TypeMem) 类型；以及 TypeFlags，也就是 CPU 的标志位类型。

这里我要特别讲一下**内存类型**。内存类型代表的是全局的内存状态。如果一个操作码带有一个内存类型的参数，那就意味着该操作码依赖该内存状态。如果一个操作码的类型是内存类型，则意味着它会影响内存状态。

SSA 的介绍文档中有一个例子，能帮助你理解内存类型的用法。

在这个例子中，程序首先会向地址 a 写入 3 这个值。这个时候，内存状态就修改了（从 v1 到了 v10）。接着，把地址 a 的值写入地址 b，内存状态又发生了一次修改。在 IR 中，第二行代码依赖第一行代码的内存状态（v10），因此就导致这行代码只能出现在定义了 v10 之后。

复制代码

```
// *a = 3        //向a地址写入3
// *b = *a       //向b地址写入a的值
v10 = Store <mem> {int} v6 v8 v1
v14 = Store <mem> {int} v7 v8 v10
```

这里你需要注意，对内存的读和写（各种 IR 一般都是使用 Load 和 Store 这两个词汇）是一类比较特殊的指令。其他的 Value，我们都可以认为它们是在寄存器中的，是计算过程中的临时变量，所以它们在代码中的顺序只受数据流中依赖关系的制约。而一旦中间有读写内存的操作，那么代码顺序就会受到一定的限制。

我们可以跟在Graal 编译器中学到的知识印证一下。当你读写一个 Java 对象的属性的时候，也会涉及内存的读写，这些操作对应的 IR 节点，在顺序上也是受到限制的，我们把它们叫做固定节点。

此外，Value 结构中还包含了两个辅助信息字段：AuxInt 和 Aux。AuxInt 是一个整型值，比如，在使用 Const64 指令中，AuxInt 保存了常量的值；而 Aux 则可能是个复杂的结构体，用来保存每个操作码的个性化的信息。

在 IR 中你还能看到基本块（Block），这是第二个重要的数据结构。Go 编译器中的基本块有三种：简单（Plain）基本块，它只有一个后继基本块；退出（Exit）基本块，它的最后一个指令是一个返回指令；还有 if 基本块，它有一个控制值，并且它会根据该值是 true 还是 false，跳转到不同的基本块。

第三个数据结构是函数（Func）。函数是由多个基本块构成的。它必须有一个入口基本块（Entry Block），但可以有 0 到多个退出基本块，就像一个 Go 函数允许包含多个 Return 语句一样。

现在，你已经知道了 Go 的 IR 的关键概念和相关的数据结构了。Go 的 IR 在运行时就是保存在 Value、Block、Func 等内存结构中，就像 AST 一样。它不像 LLVM 的 bitcode 还有文本格式、二进制格式，可以保存在文件中。

那么接下来，编译器就可以基于 IR，来做优化了。

基于 SSA 格式的 IR 做优化

SSA 格式的 IR 对编译器做优化很有帮助。

以死代码删除为例，Value 结构中有一个 Uses 字段，记录了它的使用数。如果它出现在另一个 Value 的操作码的参数里，或者是某个基本块的控制变量，那么使用数就会加 1；而如果 Uses 字段的值是 0，那就证明这行代码没什么用，是死代码，可以删掉。

而你应该记得，在第 7 讲中曾提到过，我们需要对一个函数的所有基本块都扫描一遍甚至多遍，才能知道某个变量的活跃性，从而决定是否可以删除掉它。那相比起来，采用 SSA 格式，可以说简单太多了。

基于这样的 IR 来做优化，就是对 IR 做很多遍（Pass）的处理。在 cmd/compile/internal/ssa/compile.go的代码里，列出了所有这些 Pass，有将近 50 个。你能看到每个处理步骤执行的是哪个优化函数，你还可以在 ssa.html 中，看到每个 Pass 之后，IR 都被做了哪些修改。

```
// list of passes for the compiler
var passes = [...]pass{
        // TODO: combine phielim and copyelim into a single pass?
        {name: "number lines", fn: numberLines, required: true},
        {name: "early phielim", fn: phielim},
        {name: "early copyelim", fn: copyelim},
        {name: "early deadcode", fn: deadcode}, // remove generated dead code to avoid doing pointless work during opt
        {name: "short circuit", fn: shortcircuit},
        {name: "decompose args", fn: decomposeArgs, required: true},
        {name: "decompose user", fn: decomposeUser, required: true},
        {name: "pre-opt deadcode", fn: deadcode},
        {name: "opt", fn: opt, required: true},                // NB: some generic rules know the name of the opt pass. TODO: sp
        {name: "zero arg cse", fn: zcse, required: true},      // required to merge OpSB values
        {name: "opt deadcode", fn: deadcode, required: true}, // remove any blocks orphaned during opt
        {name: "generic cse", fn: cse},
        {name: "phiopt", fn: phiopt},
        {name: "gcse deadcode", fn: deadcode, required: true}, // clean out after cse and phiopt
        {name: "nilcheckelim", fn: nilcheckelim},
```

Pass的名称

该Pass所使用的函数

图 3：compiler.go 中的 Pass

这些处理算法都是在cmd/compile/internal/ssa目录下。比如cse.go里面是消除公共子表达式的算法，而nilcheck.go是被用来消除冗余的 nil 检查代码。

有些算法还带了测试程序（如cse_test.go，nilcheck_test.go）。你可以去阅读一下，看看测试程序是如何构造测试数据的，并且你还可以通过 Debugger 来跟踪测试程序的执行过程，从而理解相关优化算法是如何实现的，这是一个很有效的学习方式。

另外，gc 还有一些比较简单的优化算法，它们是基于一些规则，对 IR 做一些重写（rewrite）。Go 的编译器使用了自己的一种 DSL，来描述这些重写规则：针对机器无关的操作码的重写规则，是在generic.rules文件中；而针对机器有关的操作码的重写规则是在 XXX.rules 中，比如AMD64.rules。

我们来看几个例子：在 generic.rules 中，有这样一个机器无关的优化规则，它是把 x*1 的运算优化为 x。

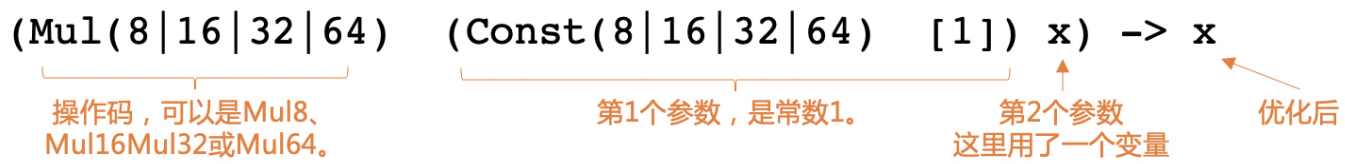


图 4：把 x*1 的运算优化为 x 的规则

在 AMD64.rules 中，有一个机器相关的优化规则，这个规则是把 MUL 指令转换为 LEA 指令，LEA 指令比 MUL 指令消耗的时钟周期更少。

```
1 (MUL(Q|L)const [ 3] x) -> (LEA(Q|L)2 x x)
```

复制代码

generic.rules 中的规则会被rulegen.go解析，并生成 Go 代码rewritegeneric.go。而 AMD64.rules 中的规则，被解析后会生成rewriteAMD64.go。其中，Lower 的过程，也就是把机器无关的操作码转换为机器相关的操作码，它也是用这种重写规则实现的。

通过 gc 这种基于规则做指令转换的方法，你应该产生一个感悟，也就是在写软件的时候，我们经常要设计自己的 DSL，让自己的软件更具灵活性。比如，gc 要增加一个新的优化功能，只需要增加一条规则就行了。我们还可以再拿 Graal 编译器印证一下。你还记得，Graal 在生成 LIR 的时候，要进行指令的选择，那些选择规则是用注解来生成的，而那些注解规则，也是一种 DSL。

好了，谈完了优化，我们继续往下看。

生成机器码

最后，编译器就可以调用 gc/ssa.go 中的genssa方法，来生成汇编码了。

在 ssa.html 的最右边一栏，就是调用 genssa 方法以后生成的汇编代码（采用的是 Go 编译器特有的格式，其中有些指令，如 PCDATA 和 FUNCDATA 是用来与垃圾收集器配合

的）。

genssa

```
       # /Users/richard/projects/mygo/src/mygo/foo.go
       00000 (3) TEXT "".Foo(SB), ABIInternal
       00001 (3) PCDATA $0, $-2
       00002 (3) PCDATA $1, $-2
       00003 (3) FUNCDATA $0, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
       00004 (3) FUNCDATA $1, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
       00005 (3) FUNCDATA $2, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
v9     00006 (+5) PCDATA $0, $0
v9     00007 (+5) PCDATA $1, $0
v9     00008 (+5) MOVQ "".a(SP), AX
v10    00009 (5) CMPQ AX, $10
v5     00010 (+10) MOVL $10, CX
v11    00011 (10) CMOVQGT AX, CX
v14    00012 (10) MOVQ CX, "".~r1+8(SP)
b1     00013 (5) RET
       00014 (?) END
```

你可能会问，**编译器在生成机器码之前，不是还要做指令选择、寄存器分配、指令排序吗？**那我们看看 gc 是如何完成这几项任务的。

寄存器分配（regalloc.go）作为一个 Pass，已经在生成机器码之前执行了。它采用的是线性扫描算法（Linear Scan Register Allocator）。

指令选择会分为两部分的工作。一部分工作，是在优化算法中已经做了一些指令选择，我们前面提到的重写规则，就蕴含了根据 IR 的模式，来生成合适的指令的规则；另一部分工作，则放到了汇编器当中。

这就是 Go 的编译器与众不同的地方。原来，gc 生成的汇编代码，是一种“伪汇编”，它是一种半抽象的汇编代码。在生成特定 CPU 的机器码的时候，它还会做一些转换，这个地方可以完成另一些指令选择的工作。

至于**指令排序**，我没看到过在 gc 编译器中的实现。我请教了谷歌的一位研究员，他给我的信息是：像 AMD64 这样的 CPU，已经能够很好地支持乱序执行了，所以指令重排序给 gc 编译器的优化工作，带来的好处很有限。

而 gc 目前没有做指令排序，还有一个原因就是，指令重排序算法的实现代价比较高，而 gc 的一个重要设计目标，就是要求编译速度要快。

扩展：Go 语言的另外两个编译器，gccgo 和 GoLLVM 都具备指令重排序功能。

课程小结

这一讲，我给你介绍了 gc 编译器的主要特点。之所以能压缩在一讲里面，是因为你已经见识了好几款编译器，渐渐地可以触类旁通、举一反三了。

在 gc 里面，你能看到很多可以借鉴的成熟实践：

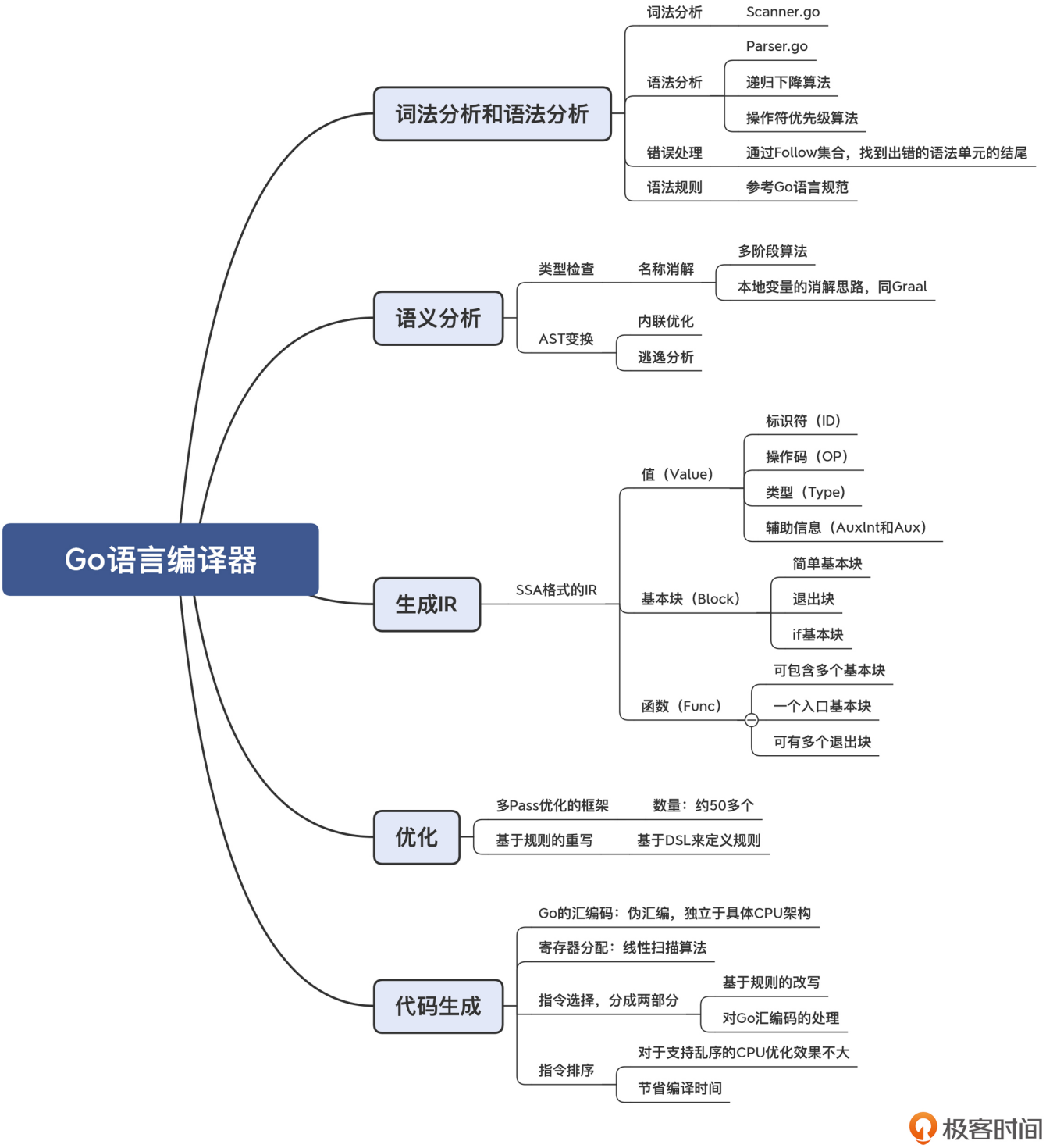
语法分析：递归下降算法，加上针对二元表达式的运算符优先级算法；

语义分析：分阶段的名称消解算法，以及对 AST 的转换；

优化：采用了 SSA 格式的 IR、控制流图（CFG）、多个 Pass 的优化框架，以及通过 DSL 支持的优化规则。

所以在这一讲的开头，我还建议你把 Go 语言的编译器作为你学习编译原理的“教学参考书”，建议你在图形化的 IDE 界面里，来跟踪调试每一个功能，这样你就能很方便去观察它的算法执行过程。

本讲的思维导图如下：



一课一思

在 gc 编译器里面，内联优化是基于 AST 去做的。那么，它为什么没有基于 SSA 格式的 IR 来做呢？这两种不同的实现会有什么差异？欢迎你在留言区发表你的看法。

参考资料

1. Introduction to the Go compiler 官方文档，介绍了 gc 的主要结构。
2. Introduction to the Go compiler’ s SSA backend 官方文档，介绍了 gc 的 SSA。

3. Go compiler internals: adding a new statement to Go - Part 1、Part2。在这两篇博客里，作者做了一个实验：如果往 Go 里面增加一条新的语法规则，需要做哪些事情。你能贯穿性地了解一个编译器的方法。
4. Go compiler: SSA optimization rules description language这篇博客，详细介绍了 gc 编译器的 SSA 优化规则描述语言的细节。
5. A Primer on Go Assembly和A Quick Guide to Go’s Assembler 。gc 编译器采用的汇编语言是它自己的一种格式，是“伪汇编”。这两篇文章中有 Go 汇编的细节。

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 23 | Julia编译器（二）：如何利用LLVM的优化和后端功能？

下一篇 25 | MySQL编译器（一）：解析一条SQL语句的执行过程

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。